Module 9

File I/O

Adapted from Absolute Java, Rose Williams, *Binghamton University*

Streams

- A *stream* is an object that enables the flow of data between a program and some I/O device or file
  - If the data flows into a program, then the stream is called an *input stream*
  - If the data flows out of a program, then the stream is called an *output stream*

Streams

- Input streams can flow from the keyboard or from a file
  - `System.in` is an input stream that connects to the keyboard

```
Scanner keyboard = new Scanner(System.in);
```

- Output streams can flow to a screen or to a file
  - `System.out` is an output stream that connects to the screen

```
System.out.println("Output stream");
```

Writing to a Text File

- The class **`PrintWriter`** is a stream class that can be used to write to a text file
  - An object of the class **`PrintWriter`** has the methods **`print`** and **`println`**
  - These are similar to the **`System.out`** methods of the same names, but are used for text file output, not screen output

Writing to a Text File

- All the file I/O classes that follow are in the package **`java.io`**, so a program that uses **`PrintWriter`** will start with a set of **`import`** statements:

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
```

- The class **`PrintWriter`** has no constructor that takes a file name as its argument
  - It uses another class, **`FileOutputStream`**, to convert a file name to an object that can be used as the argument to its (the **`PrintWriter`**) constructor

Writing to a Text File

- A stream of the class **`PrintWriter`** is created and connected to a text file for writing as follows:

```
PrintWriter outputStreamName;
outputStreamName = new PrintWriter(new
                        FileOutputStream(FileName));
```

  - The class **`FileOutputStream`** takes a string representing the file name as its argument
  - The class **`PrintWriter`** takes the anonymous **`FileOutputStream`** object as its argument

Writing to a Text File

- This produces an object of the class **`PrintWriter`** that is connected to the file ***`FileName`***
  - The process of connecting a stream to a file is called *opening the file*
  - If the file already exists, then doing this causes the old contents to be lost
  - If the file does not exist, then a new, empty file named ***`FileName`*** is created
- After doing this, the methods **`print`** and **`println`** can be used to write to the file

Writing to a Text File

- When a text file is opened in this way, a `FileNotFoundException` can be thrown
  - In this context it actually means that the file could not be created
  - This type of exception can also be thrown when a program attempts to open a file for reading and there is no such file
- It is therefore necessary to enclose this code in exception handling blocks
  - The file should be opened inside a `try` block
  - A `catch` block should catch and handle the possible exception
  - The variable that refers to the `PrintWriter` object should be declared outside the block (and initialized to `null`) so that it is not local to the block.. This is if it references elsewhere

Writing to a Text File

- When a program is finished writing to a file, it should always close the stream connected to that file

```
outputStreamName.close();
```

  - This allows the system to release any resources used to connect the stream to the file
  - If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly

Writing to a Text File

- Output streams connected to files are usually *buffered*
  - Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (*buffer*)
  - When enough data accumulates, or when the method `flush` is invoked, the buffered data is written to the file all at once
  - This is more efficient, since physical writes to a file can be slow

Writing to a Text File

- The method `close` invokes the method `flush`, thus insuring that all the data is written to the file
  - If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file
  - Also, if a program writes to a file and later reopens it to read from the same file, it will have to be closed first anyway
  - The sooner a file is closed after writing to it, the less likely it is that there will be a problem

File Names

- The rules for how file names should be formed depend on a given operating system, not Java
  - When a file name is given to a java constructor for a stream, it is just a string, not a Java identifier

  (e.g., `"C:\\Documents and Settings\\moataz\\Desktop\\fileName.txt"`)
  - Any suffix used, such as `.txt` has no special meaning to a Java program

Path Names

- When a file name is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run
- If it is not in the same directory, the full or relative path name must be given

Path Names

- The way path names are specified depends on the operating system
  - A typical UNIX path name that could be used as a file name argument is

```
"/user/sallyz/data/data.txt"
```

  - A **BufferedReader** input stream connected to this file is created as follows:

```
BufferedReader inputStream =
   new BufferedReader(new
   FileReader("/user/sallyz/data/data.txt"));
```

Path Names

- The Windows operating system specifies path names in a different way
  - A typical Windows path name is the following:

```
C:\dataFiles\goodData\data.txt
```

  - A `BufferedReader` input stream connected to this file is created as follows:

```
BufferedReader inputStream = new
   BufferedReader(new FileReader
   ("C:\\dataFiles\\goodData\\data.txt"));
```

  - Note that in Windows `\\` must be used in place of `\`, since a single backslash denotes an the beginning of an escape sequence

Path Names

- A double backslash (\\) must be used for a Windows path name enclosed in a quoted string
  - This problem does not occur with path names read in from the keyboard
- Problems with escape characters can be avoided altogether by always using UNIX conventions when writing a path name
  - A Java program will accept a path name written in either Windows or Unix format regardless of the operating system on which it is run

A File Has Two Names

- Every input file and every output file used by a program has two names:
  1. The real file name used by the operating system
  2. The name of the stream that is connected to the file
- The actual file name is used to connect to the stream
- The stream name serves as a temporary name for the file, and is the name that is primarily used within the program

IOException

- When performing file I/O there are many situations in which an exception, such as **`FileNotFoundException`**, may be thrown
- Many of these exception classes are subclasses of the class **`IOException`**
  - The class **`IOException`** is the root class for a variety of exception classes having to do with input and/or output
- These exception classes are all checked exceptions
  - Therefore, they must be caught or declared in a throws clause

Unchecked Exceptions

- In contrast, the exception classes **`NoSuchElementException`**, **`InputMismatchException`**, and **`IllegalStateException`** are all unchecked exceptions
  - Unchecked exceptions are not required to be caught or declared in a throws clause

Appending to a Text File

- To create a **`PrintWriter`** object and connect it to a text file for *appending*, a second argument, set to **`true`**, must be used in the constructor for the **`FileOutputStream`** object

```
outputStreamName = new PrintWriter(new
  FileOutputStream(FileName, true));
```

  - After this statement, the methods **`print`**, **`println`**, **`append`**, and/or **`printf`** can be used to write to the file
  - The new text will be written *after the old text* in the file

`toString` Helps with Text File Output

- If a class has a suitable `toString()` method, and `anObject` is an object of that class, then `anObject` can be used as an argument to `System.out.println`, and it will produce sensible output
- The same thing applies to the methods `print` and `println` of the class `PrintWriter`

```
outputStreamName.println(anObject);
```

Reading From a Text File Using `Scanner`

- The class `Scanner` can be used for reading from the keyboard as well as reading from a text file
  - Simply replace the argument `System.in` (to the `Scanner` constructor) with a suitable stream that is connected to the text file

```
Scanner StreamObject =
   new Scanner(new
   FileInputStream(FileName));
```

- Methods of the `Scanner` class for reading input behave the same whether reading from the keyboard or reading from a text file
  - For example, the `nextInt` and `nextLine` methods

Testing for the End of a Text File with `Scanner`

- A program that tries to read beyond the end of a file using methods of the `Scanner` class will cause an exception to be thrown
- However, instead of having to rely on an exception to signal the end of a file, the `Scanner` class provides methods such as `hasNextInt` and `hasNextLine`
  - These methods can also be used to check that the next token to be input is a suitable element of the appropriate type

Checking for the End of a Text File with **`hasNextLine`** (Part 1 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.PrintWriter;
5  import java.io.FileOutputStream;
6
7  public class HasNextLineDemo
8  {
9      public static void main(String[] args)
10     {
11         Scanner inputStream = null;
12         PrintWriter outputStream = null;
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 2 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
13          try
14          {
15             inputStream =
16                new Scanner(new FileInputStream("original.txt"));
17             outputStream = new PrintWriter(
18                             new FileOutputStream("numbered.txt"));
19          }
20          catch(FileNotFoundException e)
21          {
22             System.out.println("Problem opening files.");
23             System.exit(0);
24          }

25          String line = null;
26          int count = 0;
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 3 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
27          while (inputStream.hasNextLine( ))
28          {
29              line = inputStream.nextLine( );
30              count++;
31              outputStream.println(count + " " + line);
32          }

33          inputStream.close( );
34          outputStream.close( );
35      }

36  }
```

(continued)

Checking for the End of a Text File with `hasNextLine` (Part 4 of 4)

Display 10.4 Checking for the End of a Text File with hasNextLine

File original.txt

```
Little Miss Muffet
sat on a tuffet
eating her curves away.
Along came a spider
who sat down beside her
and said "Will you marry me?"
```

File numbered.txt (after the program is run)

```
1 Little Miss Muffet
2 sat on a tuffet
3 eating her curves away.
4 Along came a spider
5 who sat down beside her
6 and said "Will you marry me?"
```

Checking for the End of a Text File with `hasNextInt` (Part 1 of 2)

Display 10.5 Checking for the End of a Text File with hasNextInt

```
 1  import java.util.Scanner;
 2  import java.io.FileInputStream;
 3  import java.io.FileNotFoundException;

 4  public class HasNextIntDemo
 5  {
 6      public static void main(String[] args)
 7      {
 8          Scanner inputStream = null;

 9          try
10          {
11             inputStream =
12                new Scanner(new FileInputStream("data.txt"));
13          }
14          catch(FileNotFoundException e)
15          {
16             System.out.println("File data.txt was not found");
17             System.out.println("or could not be opened.");
18             System.exit(0);
19          }
```

(continued)

Checking for the End of a Text File with `hasNextInt` (Part 2 of 2)

Display 10.5 Checking for the End of a Text File with hasNextInt

```
20          int next, sum = 0;
21          while (inputStream.hasNextInt( ))
22          {
23              next = inputStream.nextInt( );
24              sum = sum + next;
25          }

26          inputStream.close( );

27          System.out.println("The sum of the numbers is " + sum);
28      }
29  }
```

File data.txt

```
1  2
3  4 hi 5
```

Reading ends when either the end of the file is reach or a token that is not an `int` is reached. So, the `5` is never read.

SCREEN OUTPUT

```
The sum of the numbers is 10
```

Reading From a Text File Using **BufferedReader**

- The class **BufferedReader** is a stream class that can be used to read from a text file
  - An object of the class **BufferedReader** has the methods **read** and **readLine**
- A program using **BufferedReader**, like one using **PrintWriter**, will start with a set of **import** statements:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
```

Reading From a Text File Using BufferedReader

- Like the classes **`PrintWriter`** and **`Scanner`**, **`BufferedReader`** has no constructor that takes a file name as its argument
  - It needs to use another class, **`FileReader`**, to convert the file name to an object that can be used as an argument to its (the **`BufferedReader`**) constructor
- A stream of the class **`BufferedReader`** is created and connected to a text file as follows:

```
BufferedReader readerObject;
readerObject = new BufferedReader(new
                    FileReader(FileName));
```

  - This opens the file for reading

Reading From a Text File

- After these statements, the methods **`read`** and **`readLIne`** can be used to read from the file
  - The **`readLine`** method is the same method used to read from the keyboard, but in this case it would read from a file
  - The **`read`** method reads a single character, and returns a value (of type **`int`**) that corresponds to the character read
  - Since the read method does not return the character itself, a type cast must be used:

```
char next = (char)
  (readerObject.read());
```

Reading From a Text File

- A program using a **`BufferedReader`** object in this way may throw two kinds of exceptions
  - An attempt to open the file may throw a **`FileNotFoundException`** (which in this case has the expected meaning)
  - An invocation of **`readLine`** may throw an **`IOException`**
  - Both of these exceptions should be handled

Reading Numbers

- Unlike the `Scanner` class, the class `BufferedReader` has no methods to read a number from a text file
  - Instead, a number must be read in as a string, and then converted to a value of the appropriate numeric type using one of the wrapper classes
  - To read in a single number on a line by itself, first use the method `readLine`, and then use `Integer.parseInt`, `Double.parseDouble`, etc. to convert the string into a number
  - If there are multiple numbers on a line, `StringTokenizer` can be used to decompose the string into tokens, and then the tokens can be converted as described above

Testing for the End of a Text File

- The method **`readLine`** of the class **`BufferedReader`** returns **`null`** when it tries to read beyond the end of a text file
  - A program can test for the end of the file by testing for the value **`null`** when using **`readLine`**
- The method **`read`** of the class **`BufferedReader`** returns **`-1`** when it tries to read beyond the end of a text file
  - A program can test for the end of the file by testing for the value **`-1`** when using **`read`**

`System.in`, `System.out`, and `System.err`

- The standard streams `System.in`, `System.out`, and `System.err` are automatically available to every Java program
  - `System.out` is used for normal screen output
  - `System.err` is used to output error messages to the screen
- The `System` class provides three methods (`setIn`, `setOut`, and `setErr`) for redirecting these standard streams:

```
public static void setIn(InputStream inStream)
public static void setOut(PrintStream outStream)
public static void setErr(PrintStream outStream)
```

`System.in`, `System.out`, and `System.err`

- Using these methods, any of the three standard streams can be redirected
  - For example, instead of appearing on the screen, error messages could be redirected to a file
- In order to redirect a standard stream, a new stream object is created
  - Like other streams created in a program, a stream object used for redirection must be closed after I/O is finished
  - Note, standard streams do not need to be closed

`System.in`, `System.out`, and `System.err`

- Redirecting `System.err`:

```
public void getInput()
{
   . . .
   PrintStream errStream = null;
   try
   {
     errStream = new PrintStream(new
           FileOuptputStream("errMessages.txt"));
     System.setErr(errStream);
     . . . //Set up input stream and read
   }
```

System.in, System.out, and System.err

```
  catch(FileNotFoundException e)
  {
    System.err.println("Input file not found");
  }
  finally
  {
    . . .
    errStream.close();
  }
}
```

Other Utilities

- The streams for sequential access to files are the ones most commonly used for file access in Java
- However, some applications require very rapid access to records in very large databases
  - These applications need to have random access to particular parts of a file
- Read/Write Files
- Binary Files—Serialization (the Serializable interface)